Join and semijoin algorithms for a multiprocessor database machine

Georges Gardarin, Patrick Valduriez

INRIA

CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 11

JOIN AND SEMI-JOIN ALGORITHMS FOR A MULTIPROCESSOR DATABASE MACHINE

Georges GARDARIN
Patrick VALDURIEZ

Avril 1982

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél: 954 90 20

JOIN AND SEMI-JOIN ALGORITHMS

FOR A MULTIPROCESSOR DATABASE MACHINE

Patrick VALDURIEZ and Georges GARDARIN

Projet SABRE, INRIA

BP 105, 78150 LE CHESNAY, France

Abstract :

This paper presents and analyzes algorithms for computing join and semi-join of relations in a multiprocessor database machine. First, a model of the multiprocessor architecture is described, incorporating I-O, CPU and message transmission cost parameters, to enable the evaluation of the execution costs of these algorithms. Then, three join algorithms are presented and compared. It is shown that, for a given configuration, each algorithm has an application domain defined by the characteristics of the operand and result relations. As semi-join operator is useful for decreasing I-O and transmission cost in a multiprocessor system, we present and compare two equi-semi-join algorithms and one inequi-semi-join algorithm. The execution costs of these algorithms are generally linear proportional to the size of the operand and result relations, and inversely proportional to the number of processors. Then, the method by joining two relations and the method by joining their semi-joins are compared. Finally it is shown that the latter method using semi-joins is generally better. The various presented algorithms are implemented in the SABRE database system; an evaluation module selects the best algorithm for performing a join according to the results presented here. A first version of the SABRE system is currently operational at INRIA.

Keywords: database machine, multiprocessor system, relational algebra, join, semi-join, filtering, performance evaluation.

ALGORITHMES DE JOINTURE ET SEMI-JOINTURE POUR UNE MACHINE BASES DE DONNEES MULTIPROCESSEUR

Patrick VALDURIEZ et Georges GARDARIN

Projet SABRE, INRIA

BP 105, 78150 LE CHESNAY, France

Résumé:

Cet article présente et analyse des algorithmes pour effectuer des jointures et semi-jointures de relations dans une machine bases de données multiprocesseur. Un modèle d'architecture multiprocesseur est décrit, avec des paramètres de coûts d'entrées-sorties, de traitement et communication, pour permettre l'évaluation des coûts d'exécution de ces algorithmes. Puis trois algorithmes de jointure sont présentés et comparés. Il est montré que, pour une configuration donnée, chaque algorithme possède un domaine d'application défini par les caractéristiques des relations opérandes et résultat. L'opérateur de semi-jointure étant utile pour diminuer le coût d'entrées- sorties et de communication dans un système multiprocesseur, nous présentons et comparons deux algorithmes d'equi-semi-jointure et un algorithme d'inequi- semi-jointure. Les coûts d'exécution de ces algorithmes sont généralement lineairement proportionnels au volume des relations opérandes et résultats, et inversement proportionnels au nombre de processeurs. La méthode de jointure de deux relations et la méthode consistant à joindre leurs semi-jointures sont alors comparées. Il est finalement montré que la dernière méthode par semi-jointures est génèralement meilleure. Les différents algorithmes présentés sont implantés dans le système de bases de données SABRE; un module d'évaluation choisit le meilleur algorithme en fonction des résultats présentés ici. Une première version du système SABRE est actuellement opérationnelle à l'INRIA.

Mots-clés : machine bases de données, système multiprocesseur, algèbre relationnelle, jointure, semi-jointure, filtrage, évaluation de performances.

1. INTRODUCTION

Several relational database machines are currently being developped (AUER80, BANE78, DEWI79, WAH80). The main objective of these projects is to answer complex queries, expressed in an assertional language against a very large data base, with better performance than conventional data base systems (CHAM81, STON76). These machines are generally realized with a set of processors executing requests in parallel. They are examples of a design approach which tends to distribute the processing power in close proximity to the data storage units. This relieves the main computer of the data processing functions.

One factor limiting the performances of relational systems is the join operation. The θ-join (CODD70), or join in short, of two operand relations R and S on attributes A from R and B from S is the result relation T obtained by concatening each tuple in R with each tuple in S, such as A θ B, where θ is an operator chosen among $=, <, \leq, \geq, >, \neq$. Join is an important operation generally needed to answer multirelation queries and can be very time consuming. A semi-join is a special case of join where the attributes of the result relation belong only to one operand relation.

Several uniprocessor algorithms have been presented and discussed in (BLAS77, GOTL75, VALD81). In such a context and without indexing, the obvious method of computing joins by nested loops has an execution time of O(n**2), for relations of cardinality n. A better method based on sorting can reduce this cost to O(n*log n) (BLAS77). A still better method based on hashing (BABB79) can further reduce the cost to O(n). However, the last method allows performing semi-joins (BERN79a).

This paper presents extensions of these algorithms to

multiprocessor systems and analyzes more precisely their performances. In section 2, the architecture model that enables us to study the algorithms is precisely described. It is derived from the SABRE database machine project (GARD81). The SABRE database machine components involved in computing joins or semi-joins are a set of filtering processors associated to the disk units, a cache memory, and a set of join processors. These processors own a local memory to store data during time processing time. They are connected together by an interconnection network. In section 3, the analysis criteria are introduced. These include the performance parameters needed for the evaluation of the execution costs (I-O, CPU and message transmission) of the algorithms. The basic assumption is that the operand relations are too large to fit into the cache memory or into the join processors' local memories. In section 4, three multiprocessor join algorithms are presented : the nested loop join algorithm (DEWI80), the sort merge join algorithm and the hashing join algorithm. Then, the execution costs of these algorithms are compared. The results mainly depend on the number of processors, the size of the operand relations, and the proportion of matching tuples in a relation. It turns out that, for a given configuration, each algorithm has an application domain defined by the characteristics of the operand and result relations.

Recent works have shown the interest of semi-join to optimize the query execution (BERN79a, BERN79b, BERN79c, CHIU80). The cost of this operation is substantially less than a join and is lineary proportional to the size of the operand and result relations. Semi-join is useful in distributed relational databases (LEBI80, ROTH80) to reduce the cost of processing queries involving binary operations, by initially selecting relevant data and thereby reducing the size of the operand relations.

We show in this paper that it is also a very useful operation to optimize join processing in a multiprocessor database machine. In section 5, two equi-semi-join algorithms are presented and compared. A third inequi-semi-join algorithm is also proposed. Their execution cost is generally proportional to the size of the operand and result relations and inversely proportional to the number of processors. In a uniprocessor environment,semi-join can be used efficiently to replace the join of relations by the join of their semi-joins (VALD81). Thus, comparisons of two join methods, one directly based on the nested loop algorithm (BORA80), and the other performing semi-joins before join are developped. These comparisons indicate the general superiority of performing joins by using semi-joins.

2. ARCHITECTURAL MODEL

The environment in which join and semi-join algorithms are studied is the SABRE database machine (GARD81). This machine is composed of a set of processors interconnected through an interconnection network. The processors exchange commands directly and data via a cache memory. We will now give a simplified model which describes the main architectural components that are useful to implement joins and semi-joins.

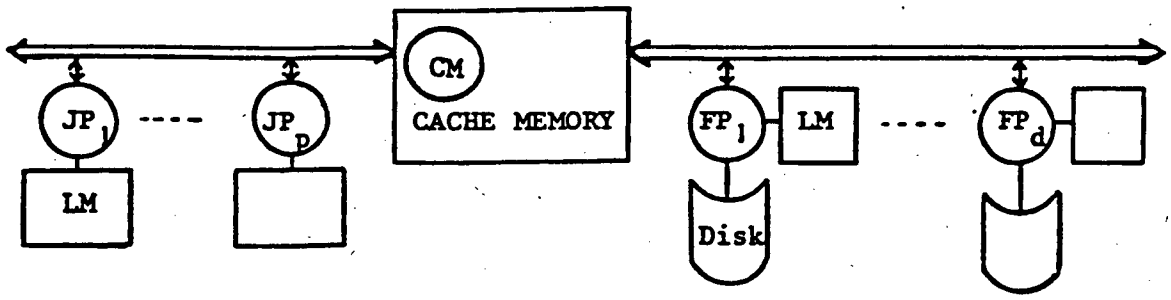
The first layer of the architecture is arranged as a set of data filtering processors (FAUD80, BANC80, ROHM81). These processors called filters perform selections (restriction and projection) on data flows coming from disks. It has been shown in (BORA81) that, if parallel readout disks would be employed, the approach processor per disk together with an efficient placement strategy gives the best results, since it avoids channel contention. Furthermore, this provides a cheap filtering power. So, in SABRE, we implement a filter per disk unit. Filters are very simple processors close to the general form of automata. They permit realization of asynchronous processing "on the fly", or in other words, filters perform a selection at the disk rotation speed in a secondary memory page , which generally corresponds to a track. A selection expression is composed of a set of references that specify the projection attributes and the selection condition. A selection condition is a logical combination of predicates of the form (Ai op C), expressed in normal conjunctive form, where Ai is an attribute of an operand relation, op an operator chosen among $<,\leqslant,=,\neq,\geqslant,>$ and C a value. The number of predicates which can be taken into account by a filter only depends on the automata characteristics , and not on the data flow and the tested expressions. In (FAUD80),

evaluations show that, for an average memory size of 64 K bits, a filter can accept up to 200 predicates.

The second layer of the architecture is a cache memory. The filters are connected with the cache memory through a multibus (INTE79). We will assume that this multibus does not provide any contention. Selected data are moved via the multibus to the cache memory, divided in pages. In Sabre, the basic processing and moving unit is the page. The cache manager should anticipate page reading and use a replacement algorithm to swap the less recently utilized pages to the disk. A filter can be associated to the cache memory to operate selection among pages already in cache.

Finally, the third layer of the architecture is a set of join processors. They are devoted to computing join of relations. Join processors send to the cache reading and writing requests to get pages of relations and to write back result pages. Pages are stored in the local memories of processors during processing. Join processors are connected with the cache memory by a specific interconnection network. The needs for this network are the ability of parallel transfers between cache and any join processor, and the capability of broadcasting a page from cache to several join processors. This kind of interconnection network is feasible (DEWI79), and different alternatives are studied. The current development of SABRE tries to be independent of the interconnection network, as it will affect the performances of the whole system. For our purpose, we will assume an ideal interconnection network providing the two needed services, and introducing no contention. These architectural components are depicted by figure 1,

where the interconnexion networks are reduced to buses, for simplicity.



JP : Join Processor
CM : Cache Manager
FP : Filtering Processor
LM : Local Memory

Figure 1 - Architecture components

3. ANALYSIS CRITERIA

In the following, we assume that a semi-join or join operation is performed on relations R and S, producing the result relation T. The following notations are needed to estimate the proposed algorithms :

m,n : number of pages in R and S, respectively,

b+1 : size of local memory (in number of pages) of each join processor (JP),

p : number of JP assigned to the operation,

c : cache memory size allocated to the join operation (in number of pages),

d : number of filters assigned to the operation,

t : number of tuples in a page, assumed to be the same for both R and S,

JS : join selectivity factor, defined by size (R join S)/(m*n),

SR : semi-join of (R by S) selectivity factor is defined by size(semi-join(R by S))/m,

SS : semi-join of (S by R) selectivity factor is defined by size(semi-join(S by R))/n,

j : average number of distinct join attribute values in a page. So, mj and nj are the total numbers of distinct join attribute values in R and S, respectively,

Nc : number of predicates accepted by a filter,

Ct : occupation factor of join attributes in a tuple, defined by size of join attribute/ size of tuple. Thus, the number of pages of the relation obtained after projecting relation S on join attributes without duplicate elimination is defined by na = n*Ct.

The presented algorithms will be compared according to their

execution cost. The basic performance parameters necessary to the evaluation of the execution cost are those detailed above. These parameters allow to measure the I-O cost, the CPU cost and the inter-processor communication cost to execute a multiprocessor algorithm. The identified parameters are dependant of the hardware capabilities, and will be fixed only for comparison purposes. All the costs will be expressed in units of time.

3-1. I-O cost

In a multiprocessor architecture, two kinds of transfers are considered as I-O operations : these are page transfers from a mass storage unit or from the cache memory to a processor's local memory. The page transfer cost from disk to cache is denoted Tdc, and includes the selection cost due to the filtering on the fly. The page transfer cost from cache to processor is denoted Tcp. The average cost of a read by a join processor is now defined by introducing the probability that the requested page is already in the cache memory. The cache manager has to maintain high such a probability by anticipating page reading, which is made easier by the fact that a referenced relation will be read entirely. This probability is assimilated to the cache hit ratio, denoted by F. Thus the average cost of a read defined by CR is :

$$CR = F*Tcp + (1-F)*(Tcp+Tdc)$$

The average cost of a write by a join processor can be evaluated in the same fashion by introducing F', the probability that there is an available page frame in the cache during the write operation. Thus the average cost of a write defined by CW is :

$$CW = F'*Tcp + (1-F')*(Tcp+Tdc)$$

3-2. Communication cost

Page moves are seen like I-O operations. Therefore, the only communication cost to take into account is the page request messages and reply messages. This message cost, denoted by M, is then added to the I-O cost. So CR is replaced by CR + M(request) + M(reply) and CW by CW + M(request) +M (reply). The control messages , like those used for processor allocation are in few number and small in size relative to the page request messages and reply messages. Thus they will be ignored.

3-3. CPU cost

Two basic operation costs are defined. O denotes the cost of a simple operation, such as comparing two attributes or computing a hashing function. The cost of moving a tuple in a page of local memory is denoted by I. With these two basic operation costs, , we can compute all other costs as follows.

join_cost : The pages to be joined may or may not be sorted. If they are unsorted, two methods for joining are possible. The first consists of internaly sorting the pages followed by a merge type join operation. The second one, which is simpler, compares each tuple in a page with each tuple in another page and is better than the first one, since it only requires comparison operations, which is cheaper than tuple movement that internal sorting needs. Thus, the cost of joining two unsorted pages, denoted by CJ0 is : CJ0 = t**2*O. The cost of joining two sorted pages by a merge type operation is defined by : CJ1 = 2*t*O.

one_page_sort_cost : The average cost for internally sorting a page of t tuples requires $t*\log_2 t$ comparisons and move operations (KNUT73). It is defined by :

$$Cs = t*\log_2 t* (O+I)$$

q_page_merge_cost : The merging of q sorted pages needs q*(q-1)*t comparison operations and q*t move operations. By adding the q page reading and writing, the q page merge cost is defined by :

$$CMq = (q*(q-1)*t*O) + (q*t*I) + (q*(CR+CW))$$

4. JOIN ALGORITHMS

4-1. The Nested Loop Join Algorithm

4-1-1. Description

This algorithm is a parallel version of the most unefficient uniprocessor join algorithm, since it composes the cartesian product of the relations. This simple algorithm is particularly well suited to parallel execution. The method is to join each page of one relation with the entire other relation. The algorithm has been described and evaluated in (BORA80) for processors having three pages of local memory, where two buffers are used to input pages and one buffer to output pages. The method is now generalized to support execution with more than three pages of local memory. The execution of this algorithm by p processors , each having (b+1) pages of local memory proceeds as follows. The smaller relation is chosen as the external one and is sequentially distributed among p join processors in blocks of (b-1) pages. Next, the second (internal) relation is broadcasted page by page to the p processors. Then, each processor joins each (b-1) page block of the external relation with the entire internal relation. For each page of the internal relation, each processor computes an internal join with (b-1) pages of the external relation, using a result buffer of one page. If the external relation does not fit into processors' local memories, the same process must be repeated for the remaining pages of the external relation.

Let us point out that, as each tuple of one relation will be compared to each tuple of the other one, this algorithm naturally supports inequi-join.

4-1-2. Evaluation of the algorithm

If the external relation (suppose R) fits into the local memories of the processors, that is $m \leqslant p*(b-1)$, each processor reads in parallel (b-1) pages of R and the execution cost of the nested loop join algorithm C(NL) is :

C(NL) = C(read (b-1) pages of R) + n* C (broadcast one page of S + join (b-1) pages of R with one page of S)

If m> p*(b-1), the process is repeated m/(p*(b-1)) times. The cost of joining (b-1) pages of R with one page of S is (b-1)*CJ0, because they are unsorted. The cost of writing the result of a two page join depends on the join selectivity factor defined by JS and is JS* ((t*I)+CW). Finally, the execution cost of the nested loop join algorithm is :

$$C(NL) = m/(p*(b-1)) * ((b-1)*CR + n*(CR + (b-1)*(CJ0 + SJ*((t*I)+CW))))$$

This cost is proportional to m/p + (m*n)/p and indicates that the smaller relation is chosen as external in order to decrease the first term. This equation also shows that the degree of parallelism is constant (i.e. the amount of work each processor performs is constant) during the entire execution of the algorithm. Let us notice ,however, that the last pass of the algorithm can use a fewer number of processors if the ratio m/(p*(b-1)) does not give an integer result.

4-2. The Sort Merge Join Algorithm

This algorithm employs a parallel sorting of the operand relations on join attributes followed by a uniprocessor merge type operation of the two sorted relations to complete the join. For simplicity, we consider the equi-join. The sorting algorithm is a parallel version of the uniprocessor ascending (or merge sort) algorithm described in (KNUT73, p.247). In (BORA80), a join algorithm is presented using a parallel binary merge sort where the arranged processors are as binary tree (PREP78). Performance analysis of this join algorithm shows that it is generally less efficient than the multiprocessor nested loop join algorithm, if the number of processors is high. This is mainly because, after a certain stage, the degree of parallelism is divided by 2 at each merge pass, so that at the last pass, one processor merges the entire relation. We propose a parallel b-way merge sorting which is more efficient in a multiprocessor environment. Moreover, if the number of processors p is less than or equal to b, the last stage only needs one pass.

4-2-1. Description of the sorting algorithm

We will breafly review the b-way merge sort algorithm. Let us suppose that there are n elements to be sorted. A run is defined as an ordered sequence of elements , so the set to be sorted contains n runs of one element. The method consists of iteratively merging b runs of K elements into a sorted run of K*b elements, starting with K=1. For pass i, each set of b runs of b**(i-1) elements is merged into a sorted run of b**i elements. Starting from i=1, the number of passes necessary to sort n elements is $\log_b n$ (KNUT73).

We will now describe the application of this method in a

multiprocessor database machine. Let us suppose we have to sort a relation of n pages which is large enough to not fit into cache memory. Let us remind that each of the p processors has a local memory of b+1 pages , where b pages are used as input pages and one as an output page. At each pass, each processor merges a set of b runs of b**(i-1) pages into a sorted run of b**i pages. The merge of b runs is done by successively reading one necessary page of each run into b input buffers and then moving ordered tuples into the output buffer, writting in cache memory when full. When read at the first pass, pages are internally sorted. To clarify the analysis, we assume that modulo (n,p) =0, i.e. that p processors divide the relation in equal parts and share exactly the same amount of work. At each merge pass, the number of runs is divided by b while the size of each run is multiplied by b, and the whole relation is read and written. For the first pass, let N the number of runs to be merged; if N is greater than p*b, one step is necessary for each of the p processors to merge b runs and this step is repeated N/(p*b) times until all the runs have been read. When N is equal to p*b, only one step is necessary and each processor merges exactly b runs of n/(p*b) pages. This pass is called the optimal stage (BORA80). The optimal stage then generates p runs of n/p pages and if p equals b, one processor can merge them in a single pass. But, as in certain configurations, p can be very much greater than b, in which case the solution is to arrange the processors as a tree of order b during the last stage, called post-optimal. The number of necessary processors is divided by b at each pass. At the last pass, one processor merges the entire relation. The number of passes for the post-optimal stage is $\log_b p$. This stage degrades the degree of parallelism. However, the result relation is sorted on the join attributes, which can be very

useful if it is asked in the query.

Joining two sorted relations is done by a uniprocessor merge type operation,where each relation is read one page at a time. The adaptation of this algorithm to the inequi-join modifies the merge type operation to be more complex and costly.

4-2-2. Evaluation of the sorting algorithm

We evaluate the execution cost of sorting a relation of n pages. At the optimal stage corresponding to the pass K, each processor produces a sorted run of b**K pages. As each of p processors has merged exactly n/p pages, the following equation is deduced :

$b^{**}K = n/p$ so $K = \log_b(n/p)$.

In counting the pass from 1, K is the number of passes until the optimal stage. During all the passes preceding the optimal stage, each processor is employed and realizes exactly n/(p*b) merge operations of b pages at each pass. The cost of a merge operation of b pages is CMb. Furthermore, each processor internally sorts n/p pages at the first pass. The execution cost of the algorithm until the optimal stage is then :

$(n/p)*Cs + \log_b(n/p)*(n/(b*p))*CMb$

The optimal stage generates p runs of n/p pages. The number of passes to merge p runs is $\log_b p$. At each pass of the post-optimal stage, the number of processors needed is divided by b while the work of each is multiplied by b. First, p/b processors perform n/p merge operations of b pages, then p/(b**2) processors perform (b*n)/p merge operations and so on until only one processor performs exactly n/b merge operations. The execution cost of the post-optimal stage is then :

$$\underbrace{(n/p + b*n/p + (b**2)*n/p + \ldots + n/b)}_{\log_b p \text{ terms}} * CMb$$

$$= ((1-b**\log_b p)/(1-b)) * (n/p) * CMb$$

The execution cost of sorting a relation of n pages is then :

$$C(\text{sort}(n)) = (n/p)*Cs + \log_b(n/p)*(n/(b*p))*CMb + ((1-b**\log_b p)/(1-b))*(n/p)*CMb$$

4-2-3. Evaluation of the multiprocessor sort merge join algorithm

The execution cost of the algorithm is the cost of sorting the relation R plus the cost of sorting the relation S plus the cost of joining the sorted relations by a merge. The uniprocessor merge type operation consists in reading the sorted relations ,joining them and writing the result relation. The cost of merging m and n pages is :

$$C(\text{merge}(m,n)) = (m+n)*CR + \max(m,n)*CJ1 + m*n*SJ*CW$$

Finally, the execution cost of the sort merge join algorithm (SM) is :

$$\begin{aligned} C(SM) = {} & (m/p)*Cs + \log_b(m/p)*(m/(b*p))*CMb \\ & + ((1-b**\log_b p)/(1-b))*(m/p)*CMb \\ & + (n/p)*Cs + \log_b(n/p)*(n/(b*p))*CMb \\ & + ((1-b**\log_b p)/(1-b))*(n/p)*CMb \\ & + (m+n)*CR + \max(m,n)*CJ1 + m*n*SJ*CW \end{aligned}$$

4-3. The Hashing Join Algorithm

4-3-1. Description

The proposed method uses hashing techniques and boolean arrays. Using boolean arrays for implementing the semi-join operation has been described by Babb (BABB79). The idea is to hash the join attribute and to use the result as an address into the boolean array. The presence of a marked bit in the array means that matching tuples exist. The power of the boolean arrays is to eliminate most of the data useless to the result. In order to support join as well as semi-join operations, the method is improved and adapted to a multiprocessor context. For simplicity, we describe the equi-join. The method proceeds in two stages, prior to which a boolean array B is initialized in cache memory. In the first stage, the smaller relation is read into the cache memory and hashed on the join attribute by the cache processor so that each tuple is written in a block of a hashed file, which is also maintained in the cache memory. The hashed file is particular, since we allow a variable number of linked pages by block and, for each block, a page frame is maintained in cache memory. This avoids to manage an overflow area. Simultaneously, for each join attribute value v, B(h(v)) is marked (set to 1), where h is a hashing function applied to the join attribute. The first stage completes when the entire relation has been hashed. In the second stage, the boolean array is broadcasted to p processors and the larger relation is sequentially distributed among p processors. Each processor uses two buffers as input pages, one buffer as the output page and one buffer to store the boolean array. So, each processor receives one page of the larger relation and does the following processing. For each tuple of the page such as the join attribute value v' satisfies $B(h(v')) = 1$, one block of the hashed file

is accessed by specifying the key v' to find the matching tuple(s). Since a block of the hashed file may contain more than one page, the block is read page by page. Each tuple of the read page of the block is then compared with v' to complete the join.

This method makes extensive use of hashing . To be applicable, the number of distinct join attribute values in the hashed relation must be significantly greater than the number of blocks and the distribution of the values should be relatively uniform. At the first stage, the choice of hashing the smaller relation is made to minimize the creation and the storage costs of the hashed file.

The classical problem with hashing is collisions. If v1 and v2 are different join attribute values, we can have h(v1)=h(v2). Since the boolean array is accessed by hashing, collisions can lead to useless access to the hashed file during the second stage. In order to reduce collisions, several hashing functions h1, h2, ... , hq can be used each associated with a boolean array B1, B2, ... ,Bq. Then for each value v, all of the corresponding bits in each Bi must be set, i.e. B1((h1(v))=1, B2(h2(v))=1, ... Bq(hq(v))=1. In (BABB79), it is shown that increasing q causes the probability of collisions to approach 0.

If the hashing function for the hashed file has the property of maintening order, i.e. given two attribute values v1 and v2, if v1 < v2 then h(v1) < h(v2), then the algorithm can support inequi-join.

4-3-2. Evaluation of the algorithm

The execution cost of the algorithm comprises the cost of hashing the smaller relation by the cache processor, the cost of distributing the larger relation among p processors and the cost of accessing the hashed file. The cost of broadcasting the boolean arrays is negligible and, thus, is ignored. We remind that c page frames are available in

cache for the join operation. So the creation of the hashed file (suppose from relation R) consists of creating m blocks if c>m or c blocks otherwise. In the first case, the hashed file could be maintained in cache memory during the entire execution of the join operation. In the latter case, pages of the same block would be linked and written on disk, and retrieved using a table of physical address.

Then, the cost of creating a hashed file is m*C (read one page of R and hash) + (m-c)* C (write one page of the hashed file on disk). The cost of reading a page of R, taking into account the ratio F, is (1-F)*Tdc. The cost of hashing a page of t tuples is t*(O+I). The cost of writing the hashed file is the cost of writing (m-c) pages since c pages are reserved in cache memory during the join execution. Furthermore, there can be available page frames in cache with the probability F'. So the cost of writing (m-c) pages from cache to disk is (m-c)*(1-F')*Tdc. Then, the execution cost of hashing a relation of m pages is :

C (hash(m)) = m*(((1-F)*Tdc) +(t*(O+I))) + (m-c)*(1-F')*Tdc

The execution cost of the second stage is the cost of reading the relation S by p processors in parallel, the cost of accessing the hashed file and the cost of writing the result relation. Each processor reads n/p pages of the relation S and for each of the t tuples accesses the boolean array, leading to the cost of (n/p)*(CR+(t*O)). The access to the hashed file is needed for each matching tuple of S. The number of matching tuples is defined by the semi-join selectivity factor SS and each block of the hashed file contains (m/c) pages. So, for each page of S, the number of pages read from the hashed file is t*SS*(m/c)*cR and this occurs (n/p) times for the entire relation S. The cost of writing the result relation of size m*n*JS in parallel by p processors is

(m*n*JS*CW)/p. Finally, the execution cost of the multiprocessor hashing join algorithm (H) is :

$$C(H) = m^* (((1-F)^*Tdc) + (t^*(O+I))) + (m-C)^*(1-F')^*Tdc + (n/p)^* ((CR+(t^*O)) + (t^*SS^*(m/c)^*CR) + (m^*JS^*CW))$$

4-4. Comparisons

For the purpose of comparisons, we fixe the values for the parameters discussed in section 3 . The chosen values according to our current implementation are the following :

O : the time to compare two attributes = 10 microseconds,

I : the time to move a tuple in a page = 250 microseconds, based on a page size of 4K bytes and a tuple length of 40 bytes,

t : the number of tuples in a page = 100,

F,F' : cache hit ratios , F=0.8, F'=0.3,

Tdc : the time of a disk-cache page transfer = 30 milliseconds,

Tcp : the time of a cache-processor page transfer = 4 milliseconds, based on an average bus bandwidth of 1 megabyte per second,

M : the cost to process a message which includes sending, transfer and receiving time is assumed to be 10 milliseconds,

Ct : occupation factor of a join attribute = 0.2, so semi-join attribute length is 8 bytes,

Using these fixed values and the previous equations, the three join algorithms are compared. For simplicity, we will denote the nested loop join algorithm as the NL algorithm , the sort merge join algorithm as the SM algorithm and the hashing join algorithm as the H algorithm.

4-4-1. Execution costs versus relation sizes

Figure 2 illustrates the behaviour of the algorithms versus relation sizes. The sizes of the two relations are assumed to be the same (m=n). The configuration comprises of 16 processors each having a local memory of 5 pages with a cache memory size (allocated to the join) of 16 pages. Three curves describe the performances of the H algorithm for three different semi-join selectivity factors (0.5, 0.1, 0.01). The

join selectivity factor is assumed to be 0.01. It appears that the SM algorithm is better than both the NL algorithm and the H algorithm (with SS>0.1), when the operand relations are large. The behaviour of the H algorithm depends essentially on SS, the semi-join selectivity factor, providing the number of matching tuples from the distributed relation with the hashed file. When SS is small (SS<0.01), the number of accesses to the hashed file is low and the H algorithm is superior. For the given configuration, when SS is less than 0.5, the H algorithm always performs better than the NL algorithm. Beyond SS 0.5, the H algorithm does not perform well (curve H0). The size of cache memory allocated to the join (denoted by C) significantly influences the execution cost of the H algorithm, because if C is high enough the hashed file can fit into cache without swapping out to disk.

The previous comparisons show that, for the given configuration, each algorithm has a domain of application where it performs better than others. Figure 3 illustrates the domains of the NL algorithm in comparison with the SM algorithm, for varying relations sizes. Similarly, figure 4 depicts the domains of the SM algorithm in comparison with the H algorithm for SS=0.1, which is a realistic coefficient.

In conclusion, one of the important result of this first comparison is that no algorithm is predominant. Therefore, a well designed database machine should implement all the algorithms and select the best one to perform a join, according to the estimated cost utilizing the previous formulas. That is the way the SABRE system performs joins.

4-4-2. Comparisons with respect to the number of processors

The configuration is essentially characterized by the number of join processors and the cache memory size. Figure 5 and 6 show the

behaviour of the algorithms for varying number of processors for small relations (figure 5) and larger relations (figure 6). The complexity of the NL algorithm is 1/p while that of the SM algorithm is 1/log p. Then, when the number of processors increases, the NL algorithm performs better than the SM algorithm. The curve of the H algorithm for SS = 0.5 would be approximatively the same as that of the NL algorithm. With a favourable semi-join selectivity factor (0.1), the H algorithm gives good results when the number of processors is high. That is because the cost of the second stage of the H algorithm is of complexity 1/p. Generally, the semi-join selectivity factor will be low and the H algorithm seems very attractive in that case. It is notable that, for the SM algorithm, after a certain number of processors, duplicating the number of processors causes a very little decrease in the execution cost. This is due to the increasing number of passes of the post-optimal stage proportional to the number of processors. In fact, the SM algorithm is better for such configurations where p=b, where only one post-optimal pass is needed. When p becomes greater than b, the post optimal stage degrades the degree of parallelism.

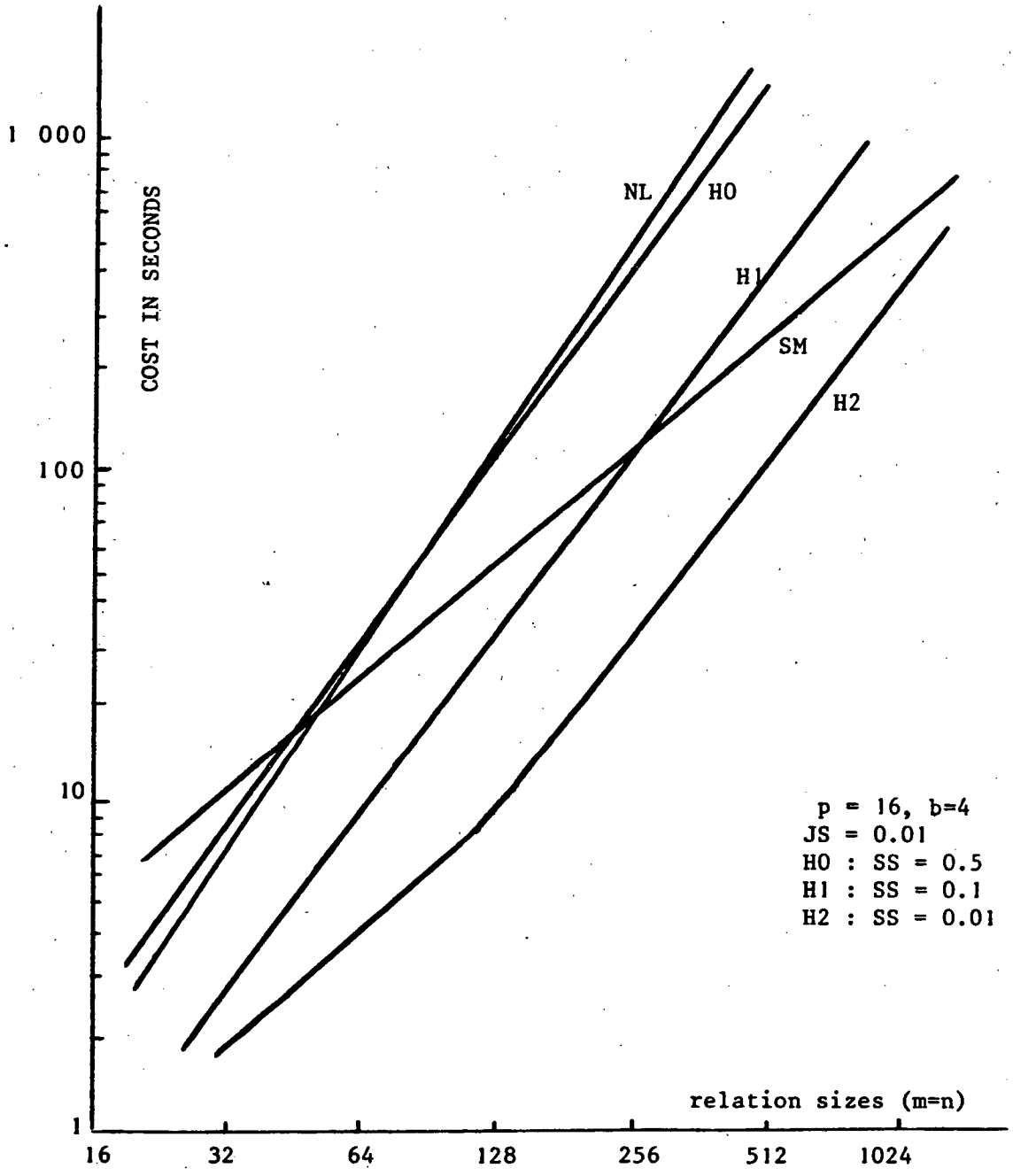


figure 2 : Execution costs of the algorithms versus relation sizes

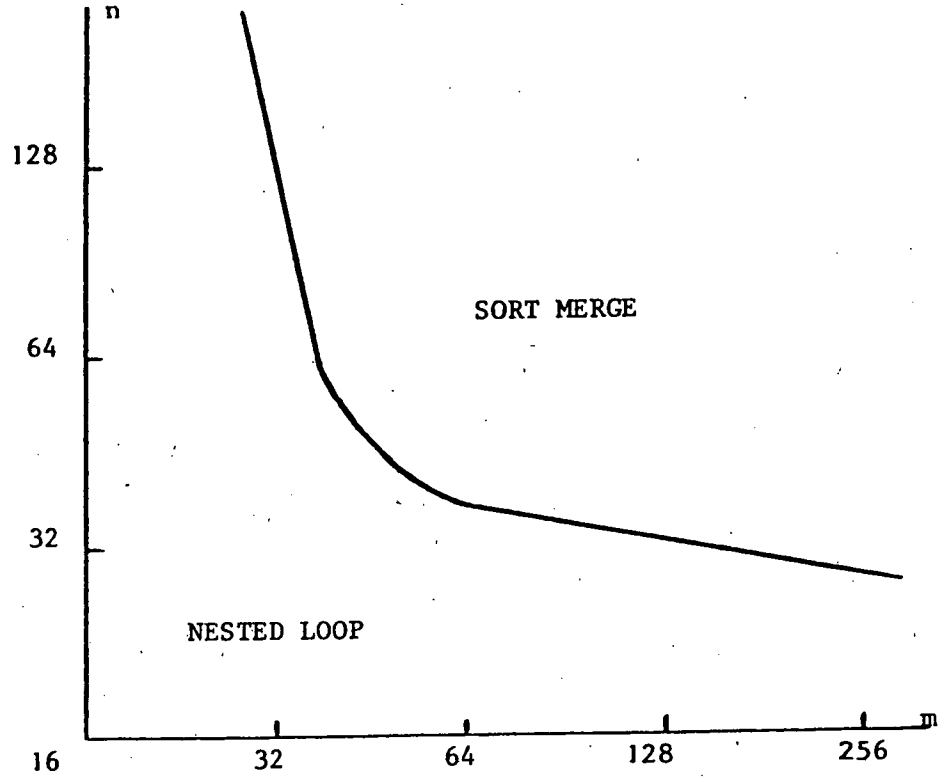


Figure 3 : Limit of the NL algorithm in comparison with the SM algorithm (p=16, b=4, JS=0.01)

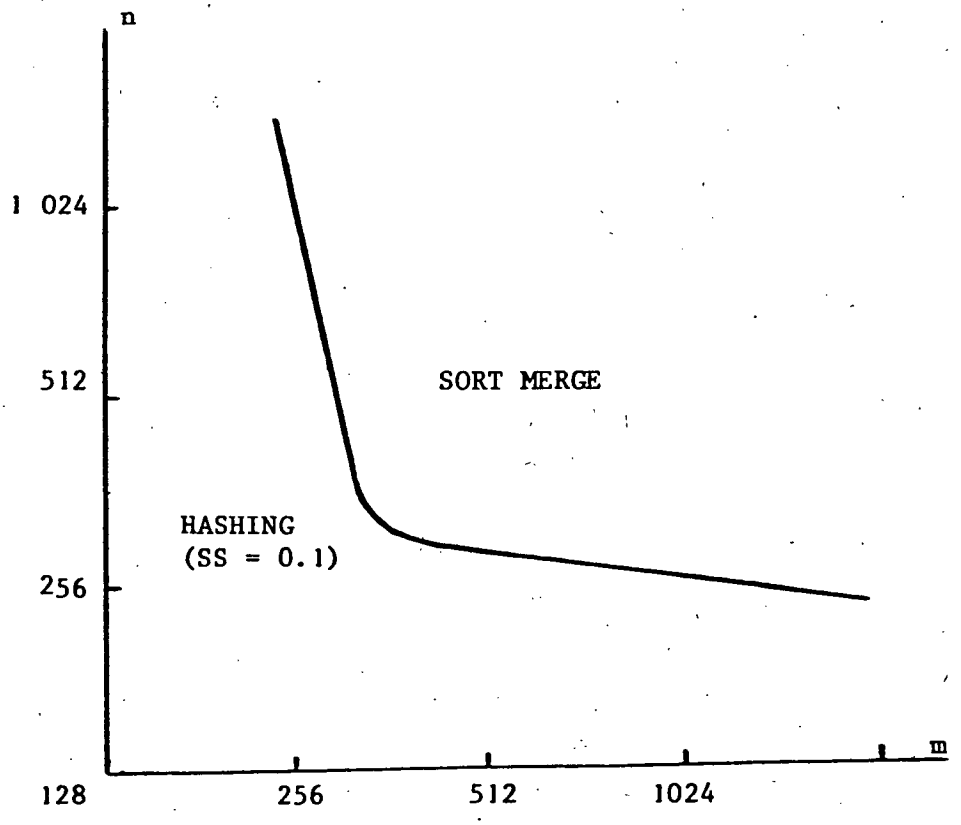


Figure 4 : Limit of the H algorithm in comparison with the SM algorithm

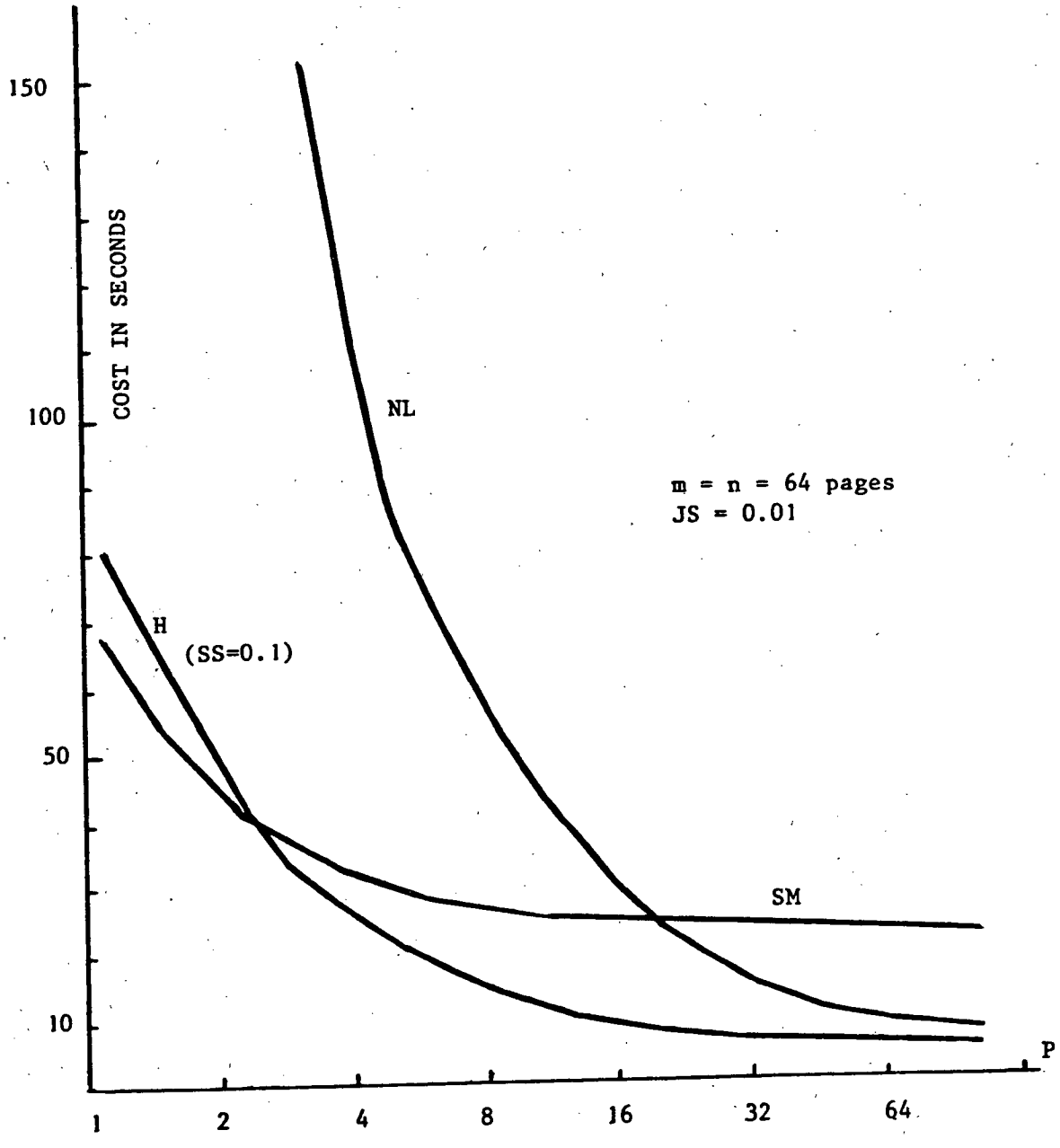


figure 5 : Execution costs of the algorithms versus the number of processors

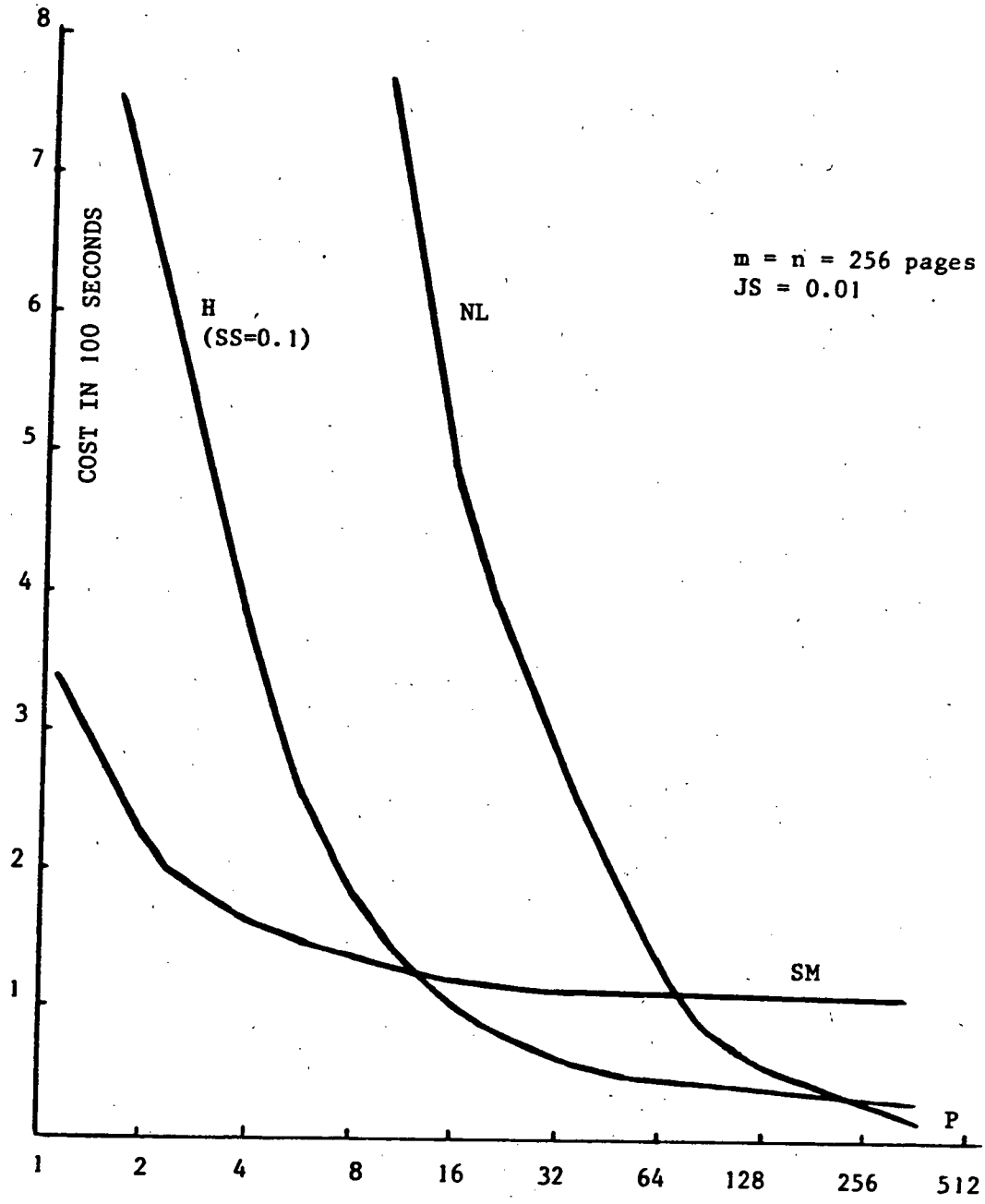


figure 6 : Execution costs of the algorithms versus the number of processors

5. SEMI-JOIN ALGORITHMS

5-1. Equi-semi-join algorithms

If the comparison operator Θ, in the join definition of section 1, is =, and if the result attributes belong to only one relation, the operation is called an equi-semi-join. In this section, we will discuss and evaluate two equi-semi-join algorithms, one using bit arrays and the other using the selection operation performed by the filtering processors.

5-1. Equi-semi-join with bit arrays

The proposed method is an extension of the uniprocessor method to perform semi-joins, as described in (BABB79). The basic version uses a bit array to aid the operation. In our parallel version, this array is replicated in join processors' local memories. The operation in a multiprocessor environment proceeds in two stages. First, semi-join source attributes are retrieved in parallel by d filtering processors , moved to the cache memory and finally compacted in pages. These pages are then distributed among p join processors. Thus, each processor reads na/p pages of the source relation S after projection --called relation S'-- , and places source attribute values in a local memory resident data structure W, in ascending order, if the value is not already in W. Simultaneously, for each source attribute value v, one bit in the bit array M is marked. The bit address is calculated by a hashing function h, where M(h(v)) is set to 1. At the end of this first stage, the bit array in each processor represents a part of the attribute values of the source relation while the union of all the bit arrays represents all the distinct attribute values. The logical union of the p arrays is then computed by one processor, which broadcasts the

result to the p-1 others. The intermediat lists (W) of each processor are merged to form a single list containing all the distinct source attribute values. This step is performed by one processor in one b-way merge pass if $p \leq b$, or in $\log_b p$ passes if $p>b$ where the number of processors is divided by b at each pass. The resultant merged list does not contain any duplicates and is broadcasted to the remaining p-1 processors. The main assumption of this method is that a processor's local memory can contain both the bit array and the list W. This assumption holds in most of the cases. If, however the list does not fit into local memory, the problem is treated like join problem studied in section 4.

The second stage of the algorithm is to distribute the target relation R (after selection by filters) among p processors. Each processor gets a page of R, and each tuple, whose join attribute value v' satisfies M(h(v'))=1, is projected into the result relation if v' is in W. This latter checking is necessary because of the possible collisions with hashing. (Note that more than one bit array can be used to decrease the probability of collisions.)

5-1-1. Evaluation of semi-join algorithm using bit arravs

The total cost of the semi-join algorithm using bit arravs is the sum of the I-O cost and the CPU cost which can be represented as

$$C\ (SJH) = C_{IO}\ (SJH) + C_{CPU}\ (SJH)$$

1) I-O cost

The I-O cost consists of the inter-processor page move cost, incurred by operand relation reading, union, merge operations plus the cost of writing the result relation. First, relation S is projected over the semi-join attributes to relation S'. This operation requires

reading S entirely from disk to cache by d filters. This costs (n/d)*Tdc. The relation S' has na pages and is read in parallel by p processors for a cost of (na/p)*CR.

We suppose that the q bit arrays in a processor's memory are contained in a page, so the union operation needs p inter-processor moves plus one more to broadcast the result. This costs (p+1)*Tcp. The merging of p attribute lists is done in $\log_b p$ passes by a b-way merge, and assuming an average list size of b/2 pages, the merge I-O cost is $(\log_b p*(b/2))*Tcp$.

The cost of the second stage of the algorithm consists of reading relation R, of size m, with a cost of (m/p)*CR and writing the result relation T, whose size depends on the semi-join selectivity factor SR, for a cost of (m/p)*SR*CW.

Finally, the I-O cost is :

$$C_{IO}(SJH) = (n/d)*Tdc + (na/p)*CR + ((p+1)+(\log_b p*(b/2)))*Tcp + (m/p)*(CR+(SR*CW))$$

2) CPU cost

The CPU cost of the semi-join algorithm using bit arrays is now calculated. During the first stage, for each read page of the relation S' (containing t/Ct attribute values) a hashing function is applied to each value which is then inserted in list W. We define the cost of inserting an element in W by Iw. Then, the CPU cost of this stage is

(na/p)*(t/Ct)*(O+Iw)

The merge cost of the p lists is denoted by Cm. The union of the p arrays needs p comparisons, i.e. p*O time units.

During the second stage, for each read page of R, a hashing function is applied to the t attribute values. Furthermore, for the t*SR tuples held in a page, W is accessed to confirm the absence of a

collision. The list W contains nj attribute values in sorted order. Thus, binary search (KNUT73, p406) can be applied to W, for a cost of $(\log_2 nj)*O$ per tuple. The move cost of the tuples in the result page is SR*I for each read page of R. The cost of the second stage is therefore $(m/p)*((t+(t*SR*\log_2 nj))*O + SR*I)$.

Thus, the total CPU cost of this algorithm is :

$$C_{CPU}(SJH) = (na/p)*(t/Ct)*(O+Iw) + p*O + Cm + (m/p)*((t+(t*SR*\log_2 nj))*O + SR*I)$$

5-1-2. Equi-semi-join using selection

This method fully depends on the selection devices offered by the filtering processors (FAUD80) to compute the equi-semi-join. The operation proceeds in two stages. The first stage is similar to the previous algorithm's first stage, however bit arrays are not used. The second stage realizes the semi-join by restricting the target relation to the tuples whose semi-join attribute value is in the list W. Only one processor is needed to initiate the operation. From the merged list W, this processor constructs selection conditions in the target relation by constructing a restriction predicate. This predicate is a disjunction of clauses of the form (A=Bi), where A is the semi-join attribute of R and Bi are the distinct attribute values in W. If the number of Bi's is greater than Nc, which is the number of predicates accepted by a filter, several selection operations are involved. The number of operations is then Sup(nj/Nc). Since every selection operation requires reading R entirely, the performance of this algorithm depends on the filter processing capabilities. We should remind that a simple filter could accept a high number of predicates.

5-1-2-1. Evaluation of the semi-join algorithm by selection

The cost of this algorithm also consists of the I-O and the CPU costs.

1) I-O cost :

The algorithm proceeds in two stages. The first stage,similar to the previous algorithm, is to project relation S over semi-join attributes into relation S', which costs (n/d)*Tdc and to read the relation S' in parallel by p processors for a cost of (na/p)*CR. The merge operation costs the same as the merge in the previous algorithm, which is $(\log_b p*(b/2))$ *Tcp.

The second stage is the selection performed in parallel by d filters on relation R (of size R). One selection on the entire relation costs (m/d)*Tdc and may have to be repeated nj/Nc times if nj is greater than Nc. It should be noted that the selection cost includes the write cost of the result relation in cache. Then, the cost of this second stage is (nj/Nc)*((m/d)*Tdc). Finally, the I-O cost of the algorithm is

$$C_{IO}(SJS) = (n/d)*Tdc + (na/p)*CR + (\log_b p*(b/2))*Tcp + (nj/Nc)*((m/d)*Tdc)$$

2) CPU cost

The CPU cost is incurred by the insertions in the list W containing the semi-join source attribute values. This cost is the same as in the previous algorithm , which is (na/p)*(t/Ct)*Iw. Also, the merge cost of p lists is noted Cm.

The CPU cost of the second stage is due to the initialization of the selection operations. The cost of constructing selection predicates with nj attributes in list W is nj*O. The CPU cost of the algorithm is

then

$$C_{CPU}(SJS) = (na/p)*(t/Ct)*Iw + Cm + (nj*O)$$

5-1-3. Comparisons

This section presents performance comparisons of the two equi-semi-join algorithms using the cost formulas developed in the previous sections. Results are shown in figures 7 and 8. Assumptions concerning the processor capabilities, as parameterized by the costs defined in section 3 are the same as these of section 4.

Figure 7 illustrates the behaviour of the algorithms with varying relation sizes. The sizes of both the source and target relations are assumed to be the same (m=n). The configuration chosen to represent a multiprocessor architecture is composed of 16 filters and 16 join processors, each owning 5 pages of local memory. A filter is assumed to accept up to 100 predicates. The stage of reading the projected relation S' and merging the lists W is the same for both the algorithms, therefore parameters Ct, Cm and Iw do not influence the performance. Figure 7 consists of 4 curves. The two curves representing the cost of the algorithm using bit arrays (SJH) vary lineary according to the size of relations (n) and depend on two different semi-join selectivity factors. The selectivity factor slightly affects the write cost of the result. The two cost curves of the algorithm using selection (SJS) are dependent on two distinct nj values, and show that this algorithm is heavily influenced by the ratio Sup (nj/Nc), which is the number of needed selection operations. This can be seen by the stair form of the curves. This algorithm is better when there are a few distinct attribute values. More generally,the algorithm by selection is better than the algorithm using bit arrays when the approximate expression

$$(m/p)*(CW+(SR*CW)) - (Sup(nj/Nc) *((m/d)*Tdc)$$

is strictly positive.

The number of join processors p divides the cost of the algorithm using bit arrays as the number of filters d divides the cost of the algorithm by selection.

Figure 8 represents the execution costs as a function of the number of processors, assuming that p=d. Performances of these algorithms are proportional to the number of processors, and the difference between the two curves is also due to the ratio (nj/Nc). Two curves depict the performances of the algorithm by selection (nj=n, nj=4n). The curve for the selection algorithm for nj=n is the same as the curve for the algorithm using bit arrays. For greater values of nj, the algorithm by selection is worse. The performance of the algorithm by selection depends on the number of filters and their capabilities.

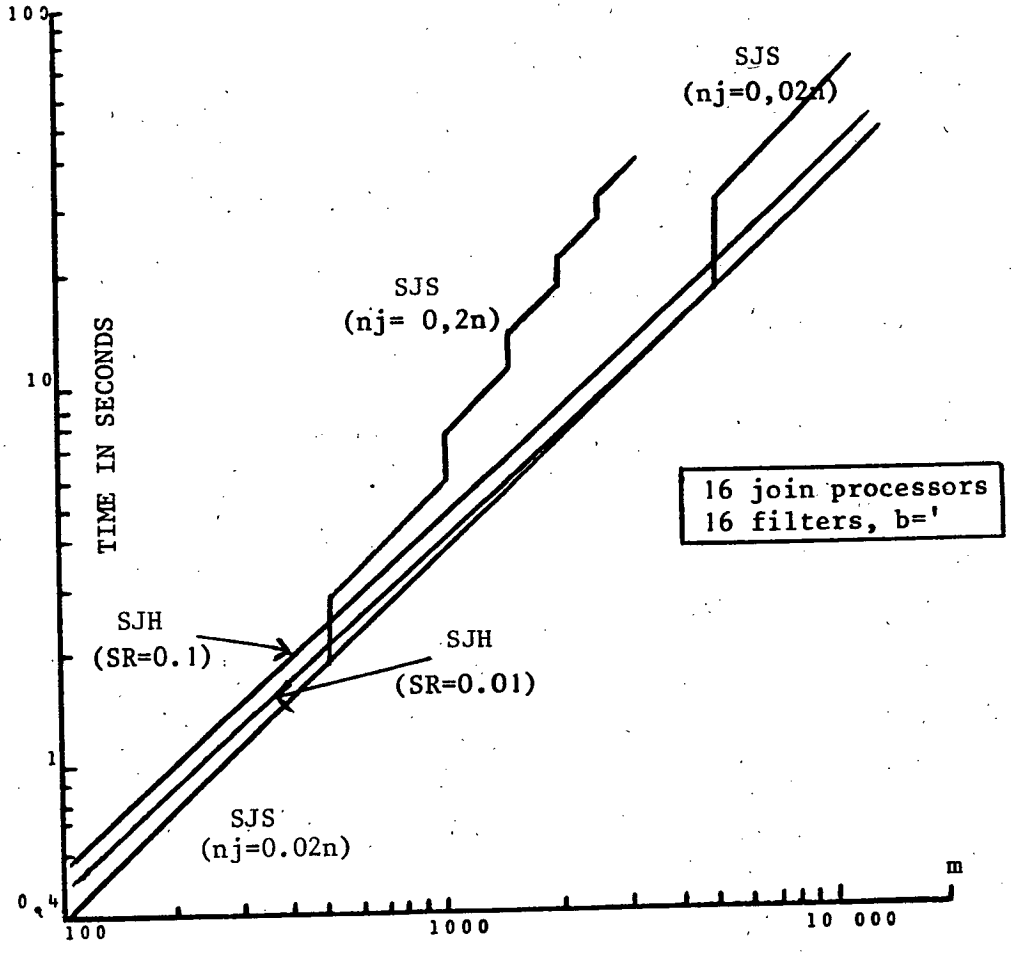


Figure 7 : Execution costs versus relation sizes

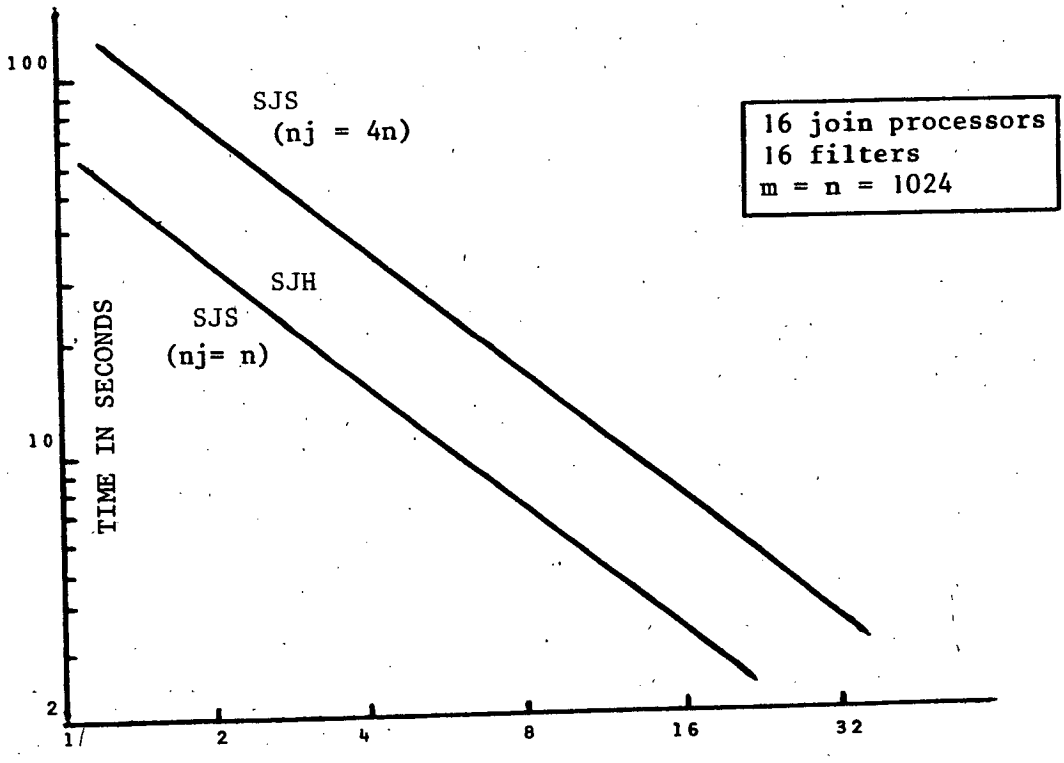


Figure 8 : Exécution costs versus the number of processors

5-2. Inequi-Semi-Join Algorithms

Inequi-semi-join is the case when the comparison operator is one of $<, \leqslant, \geqslant, >$. In this section, we will propose a simple algorithm to perform inequi-semi-join in a multiprocessor database machine. The method is based upon the following observation : the inequi-semi-join consists of selecting those tuples r of the target relation such that their semi-join attributes satisfy "r.A op X", where X is either the minimum attribute value in the source relation, if op is $\geqslant$ or $>$, or the maximum value if op is $\leqslant$ or $<$. The search for a maximum or minimum value in a data set is an easy function supported by the filtering processors. Each read attribute value is compared by the filter to a maximum or minimum current value in its local memory and replaces the current value if the comparison holds. Then, d filters get partial results, which will be integrated into one result by a processor which can be the filter controler. The inequi-semi-join is finally performed by a restriction in parallel ,by d filters among tuples in the target relation which satisfy A op X. Furthermore, this algorithm has the advantage of not using the cache.

5-2-1. Evaluation

First, the CPU cost of this algorithm is negligible, since it only consists of two function calls. The I-O cost is the reading cost of the source relation ,of size n, necessary to calculate the minimum or maximum in parallel by d filters, plus the reading cost of the target relation ,of size m, to apply the selection. The result relation is written directly in cache, so the write cost is included in the selection cost. Then, the execution cost of this algorithm is :

$$C\ (ISJ) = (\ (n/d) + (m/d)\ ) * Tdc$$

It is assumed that the cache manager always maintains available pages during the operation, in order to not slow down the filters by waiting.

6. JOIN USING SEMI-JOIN

The previously proposed semi-join algorithms are computed in a time lineary proportional to the size of the operands. The main advantage of semi-join operator is to reduce the operand relation to only the tuples participating in join. Therefore, it may sometimes be advantageous to replace the join of two relations by the join of two semi-joins. In order to compare the two alternatives (join or semi-join), we have chosen the multiprocessor nested loop join algorithm presented in section 4-1, since it is simple and generally good with a high degree of real parallelism.

The nested loop join of the initial relations is denoted by NL and is then compared to the nested loop of the semi-joins of each relation by the other, denoted NLSJ. The semi-join algorithm using bit arrays is chosen for comparisons, since it is more general than the semi-join algorithm using selection and more frequently used than the inequi-semi-join algorithm. The condition that has to be satisfied to apply this method is that the entire list of distinct join attribute values fit into local memory. When this condition cannot be satisfied, semi-join is solved like join. Therefore, it is assumed that the condition is true for the comparisons of this section.

Comparisons of the two methods NL and NLSJ are illustrated in figures 9, 10, 11 and 12. The basic configuration is the same as in section 4-4. Figure 9 presents the performances of the two methods according to the ratio JS' = size(R join S) / (m'*n'), where m' and n'

are the sizes of the semi-joins of R by S and of S by R, respectively. For each method, three join selectivity factors are considered (JS=0.16, 0.016, 0.5) while the relation sizes are kept high (m=n=1024 pages). The parameter JS' significantly influences the algorithm using semi-joins. As JS' approaches 0, the size of the join result approaches the size of the bigest semi-join result; as JS' approaches 1, the size of the join result approaches the size of the cartesian product of the semi-join results. In the latter case, semi-join reduces the size of the initial relations significantly. The method using semi-join is generally better than the nested loop method and becomes very superior when the join selectivity factor decreases. Figure 10 illustrates the domains of the two methods for varying JS and JS' and shows that the NL method is only competitive when the ratio (JS/JS') approaches 1. In that case, performing semi-join does not reduce the sizes of the operand relations.

Figure 11 illustrates the performances of the two methods according to the size of the relations(assumed to be equal) for a join selectivity factor of 0.016 and the ratio JS' equal to 0.4. The curves show the evident superiority of the method using semi-joins. This is because the semi-join cost is negligible compared to the join cost, and the operand relations to be joined will be smaller with semi-join method than with join method.

The performance of the algorithms with increasing number of processors as depicted in figure 12, still indicates that the performance of the semi-join method is superior for a join selectivity factor of 0.016, which is representative. The difference between the two methods may be reduced a little by increasing that factor.

The comparisons show that preceding the application of the nested

loop join algorithm by a semi-join of the operand relations generally decreases the total cost of the join, which confirms the interest for the semi-join operator. Similar comparisons were made , based on the sort merge join algorithm and the hashing join algorithm, and similar result were found.

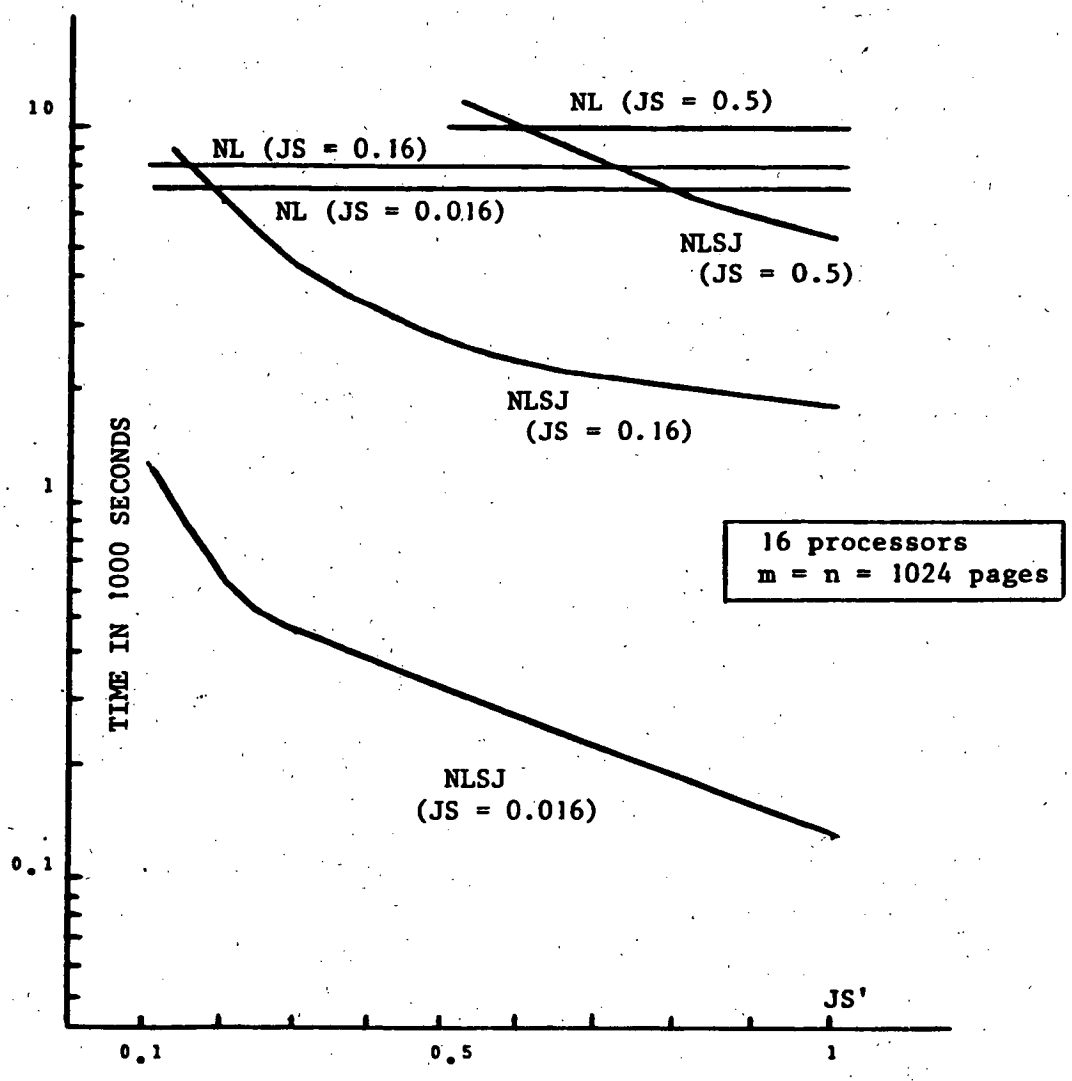


figure 9 : Comparisons of the cost of the nested loop join method (NL) and the join method using semi-join (NLSJ) versus JS´

JS´ = size (r join S) /
(size(R semi-join S) * size(S semi-join R))

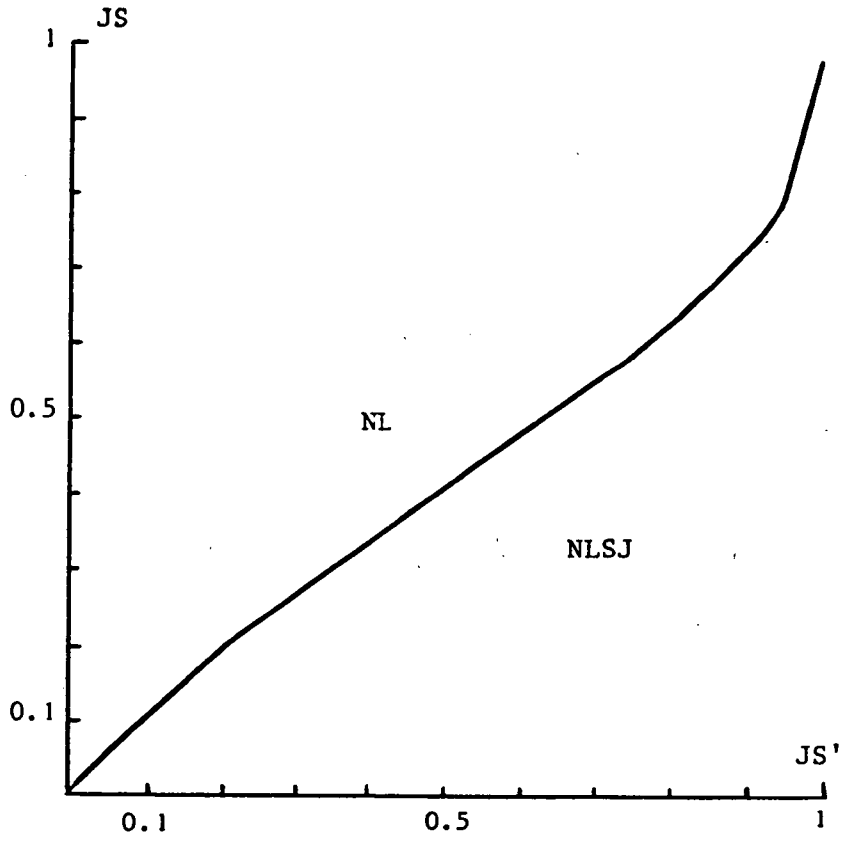


figure 10 : Limit of the methods for varying JS and JS'
(p=16, b=4, m=n=1024)

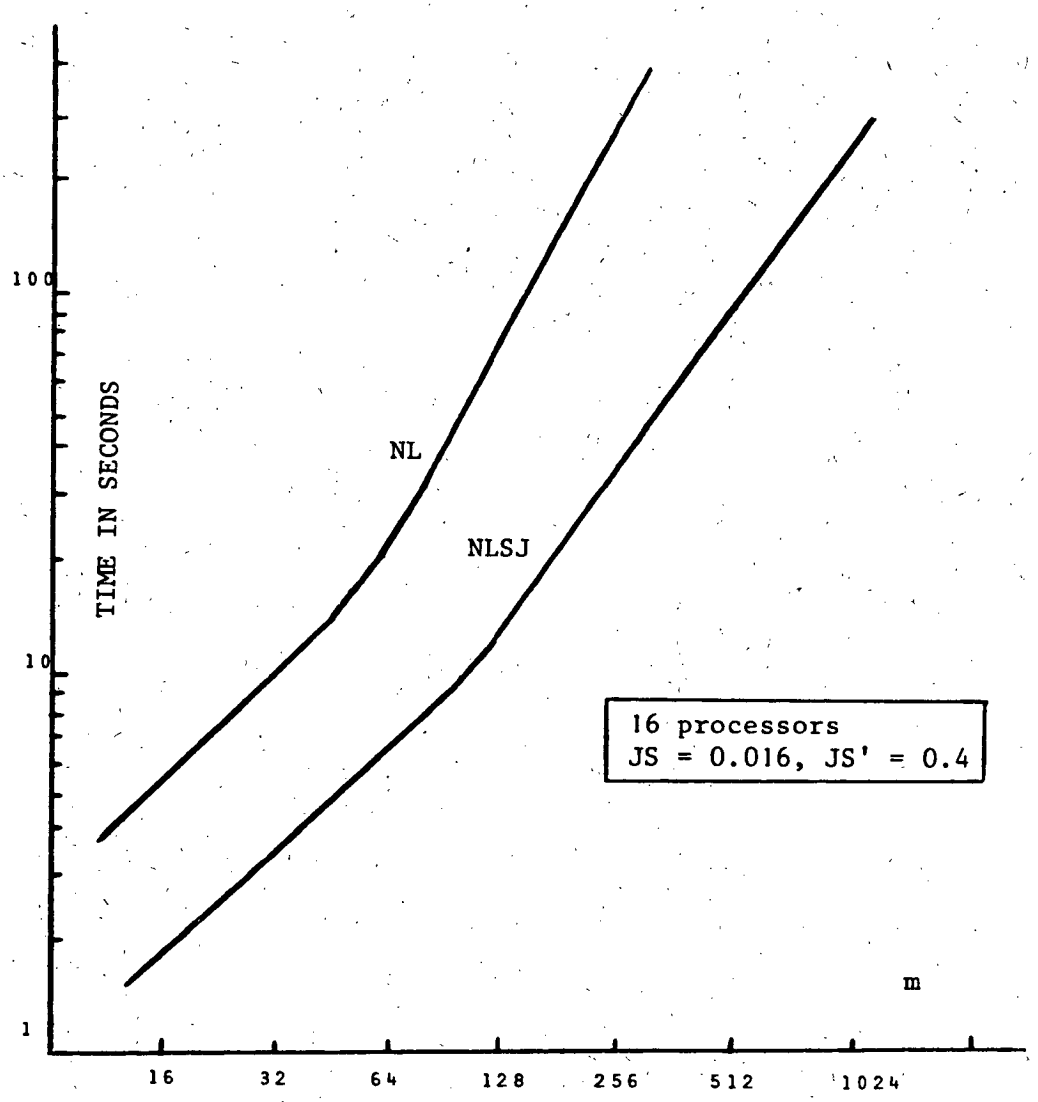


figure 11 : Comparisons of the cost of the algorithms versus relation sizes

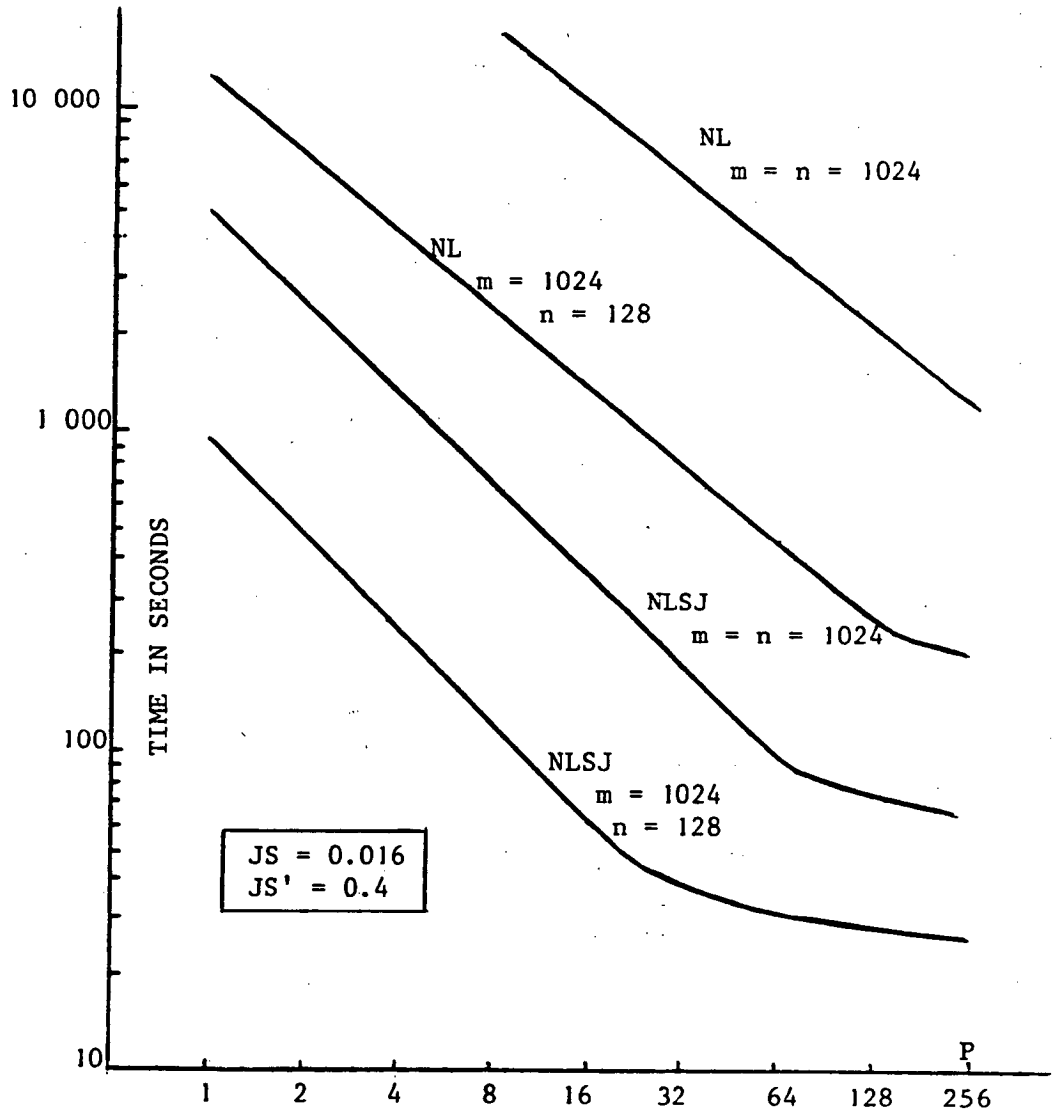


figure 12 : Comparisons of the costs of the algorithms versus the number of processors

8. CONCLUSION

In this paper, algorithms for computing joins and semi-joins in a multiprocessor database machine have been presented and analyzed. First, the architectural components useful to compute joins in the SABRE database machine, and the cost parameters including I-O, CPU and communication costs have been described. Then, three join algorithms have been presented and compared according to their execution costs. The nested loop join algorithm is the simplest and its execution cost is inversely proportional to the number of processors. It is the best when the number of processors is very high. Furthermore, since each tuple is compared with every other tuple, it easily supports inequi-join. The sort merge join algorithm is more complex and performs better as the operand relations become large. For a certain configuration, the addition of new processors causes very little increase in the performances. But it has the advantage of using another useful operator (sorting) and producing a result relation sorted on the join attributes. The hashing join algorithm is better when the number of matching tuples in the larger relation is small. In that case, using bit arrays results in avoiding a lot of useless access to the hashed file. It is shown that, for a given configuration, each of the algorithms can be the best one according to the characteristics of the operand and result relations. Since join is very important in relational systems, a well designed database machine would implement different algorithms and , for each join, choose the best algorithm by computing their execution costs. All these algorithms could be easily implemented on general purpose microprocessors without requiring special and costly hardware.

Focusing on the inter-processor transfers in a multiprocessor database machine and considering that semi-join reduces the transmission

costs, we presented parallel algorithms for semi-join operations. Two multiprocessor equi-semi-join algorithms were analysed. The algorithm using bit arrays gives good results ; its execution cost is proportional to na/p + m/p + r/p, where m is the number of pages of the target relation, na the number of pages containing the projected source semi-join attributes, r the number of pages of the result relation and p the number of processors. But the main assumption in its application is that the list of distinct semi-join attribute values should fit into processor's local memory. A second equi-semi-join algorithm using selection is proposed. It depends upon the parallel filtering power offered by the selection processors assigned to the disks. It is superior to the first algorithm when the semi-join can be computed in only one pass, that is when the number of distinct semi-join attribute values is less than the number of admitted predicates in a filter. Otherwise, the algorithm using bit arrays is better.

An inequi-semi-join algorithm is also proposed. It uses the maximum and minimum functions provided by the filter processors and does not need any place in cache during the execution except for writing the final result. This algorithm has an execution cost proportional to m/d + n/d, where m and n are the size of the operand relations and d the number of filters associated to the operation.

Comparisons are then done between the two join strategies. The method to join two operand relations and the method to join the result relations of semi-joins are compared ,using the nested loop join algorithm. Finally it is shown that the method by semi-joins is generally better, which indicates the implementation of this operator in such a database machine is useful.

The results can be extended in several directions. First, the

presented algorithms can be optimized. For example, the post optimal stage of the multiprocessor b-way merge which degrades the degree of parallelism can be improved by using another more complex method. Second, oters methods based on index or preevaluated joins can be found. These index would be organized in such a fashion as to facilitate parallel execution. The performances of the methods would essentially depend on the placement strategies of relations on disk units. An original multi-attribute clustering technique (KARL81) will be used in SABRE and will improve join executions.

ACKNOWLEDGMENTS

The authors wish to thank P.Bernadat, P.Faudemay, K.Karlsson, Y.Viemont and the other members of the SABRE project, for their helpful discussions. They also wish to acknowledge many precious comments of T.Ozsu, from the Ohio State University.

REFERENCES :

AUER80 H. Auer et al. "R.D.B.M : a Relational Data Base Machine" Internal Report, Technisch Universitat Braunchweig, nr8005, june 1980.

BABB79 E. Babb "Implementing a Relational Data Base by Means of Specialized Hardware" ACM Transactions on Database Systems, vol.4, nol, march 1979, pp 1-29.

BANC80 F. Bancilhon, M. Scholl "Design of a Back-End Processor for a Database Machine" Proceedings of the ACM-SIGMOD, Santa Monica, California, may 1980.

BANE78 J. Banerjee, D.K. Hsiao, and R.I. Baum "Concepts and Capabilities of a Database Computer" ACM Transactions on Database Systems, vol.3, no.4, december 1978, pp 347-384.

BERN79a P.A. Bernstein, D.M. Chiu "Using Semi-joins to Solve Relational Queries" JACM 80.

BERN79b P.A. Bernstein, N. Goodman "Full Reducers for Relational Queries Using Multi-Attribute Semi-joins" Proc. 1979 NBS Symp. On Comp. Network, dec 1979.

BERN79c P.A. Bernstein, N. Goodman "Inequality Semi-joins" Technical Report, CCA-79-28, dec 1979.

BLAS77 M.W. Blasgen and K.P. Eswaran "Storage and Access in Relational Data Bases" IBM System Journal, vol.16, no.4, 1977, pp 363-378.

BORA80 H. Boral, D.J. DeWitt, D. Friedland, W.K. Wilkinson "Parallel Algorithms for the Execution of Relational Operations" Technical Report 402, Computer Science Department, University of Wisconsin, Madison, janvuary 1980.

BORA81 H. Boral, D.J. DeWitt, W.K. Wilkinson "Performances Evaluation of Associative Disk Designs" 6th Workshop on Architecture for Non Numeric Processing, Hyeres, France, june 1981.

CHAM81 D.D. CHAMBERLIN, A.M. GILBERT, R.A. YOST "A History of System-R and SQL/ Data System" 7th Int. Conf. on Very Large Data Bases, september 1981, Cannes, France.

CHIU80 D.M. Chiu, Y.C. Ho "A Methodology For Interpreting Tree Queries into Optimal Semi-Join Expressions" ACM Trans. On Database Syst., 18, 4 , december 1980, 169-178.

CODD70 E.F. Codd "A Relational Model of Data for Large Shared Data Banks" CACM 13, june 1970, pp 377-387.

DEWI79 D.J. DeWitt "Query Execution in DIRECT" Proceedings of the ACM-SIGMOD 1979, International Conference of Management of Data, may 1979,pp 13-22.

DEWI80 H. Boral, D.J. DeWitt " Design Consideration for Data-Flow Database Machines" Proceedings of the ACM-SIGMOD, Conference of Management of Data, Los Angeles, 1980, pp 94-104.

FAUD80 P. Faudemay "Sur une Nouvelle Classe de Filtres Multi-Expressions" Journees Machines Bases de Donnees, Sophia Antipolis, septembre 1980, INRIA.

GARD81 G. Gardarin "An Introduction to SABRE : a Multi-Microprocessor Database Machine" 6th Workshop on Computer Architecture for Non Numeric Processing, Hyeres, France, june 1981.

GOTL75 L.R. Gotlieb "Computing Joins of Relations" Proceedings of the ACM-SIGMOD, Conference of Management of Data, San Jose, California, may 1975, pp 55-63.

INTE79 INTEL "Les Concepts Systemes d'Intel : le Multibus et ses Signaux" E.A.I.263, A. Sabatier, Intel-France, fevrier 1979.

KARL81 K. Karlsson "Reduced Cover-Trees and their Application in the SABRE Access Path Model" 7th Int. Conf. on Very Large Data Bases, september 1981, Cannes, France.

KNUT73 D.E. Knuth ,"The Art of Computers Programming" , Vol.3, Sorting and searching, Reading, MA : Addison-Wesley, 1973.

LEBI80 J. LeBihan et al. "SIRIUS : A French Nationwide Project on Distributed Data Bases" 6th Int. Conf. on Very Large Data Bases, Montreal, 1980, Proc. IEEE.

OZKA77 E.A. Ozkarahan, S.A. Schuster, K.C. Sevcik "Performance Evaluation of a Relational Associative Processor" ACM Trans. On Database Syst., june 1977, 175-196.

PREP78 F.P. Preparata "New Parallel-Sorting Schemes" IEEE Transactions on Computers, vol.C-27, july 1978.

ROTH80 J.B. Rothnie et al. "SDD1 : A System for Distributed Databases" ACM Trans. On Database Syst., march 1980, 1-17.

ROHM81 J. Rohmer "Machines et Langages pour Traiter les Ensembles de Donnees (Textes, Tableaux, Fichiers)" These d'Etat, Grenoble, 18 decembre 1980.

STON76 M. Stonebraker , E. Wong , and P. Kreps "The Design and Implementation of INGRES" ACM Transactions on Data Base Systems, vol.1, no.3, september 1976.

VALD81 P. Valduriez "Algorithmes de Jointures de Relations" Colloque "Les Bases de Donnees", AFCET, Tunis, avril 1981.

WAH 80 B.W. Wah and S.B. Yao "DIALOGUE : a Distributed-Processor Organization for a Database Machine" National Computer Conference, 1980, pp 243-253,.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique